Science of Computer Programming 15 (1990) 171-183
North-Holland

A SYSTOLIC DESIGN FOR ACCEPTORS OF REGULAR LANGUAGES

Anne KALDEWAIJ and Gerard ZWAAN

Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, Netherlands

Revised June 1990

Abstract. In this paper we present a derivation method for networks of systolic cells. The method is calculational and can be applied for specifications that are data-independent, i.e. the order of communications does not depend on the values communicated. The techniques used in this style of parallel programming are comparable with techniques used in sequential programming. The presentation is done by means of an example: the derivation of a set of building blocks for the construction of acceptors for regular languages.

1. Introduction

In this paper we derive a set of building blocks (systolic cells) that can be used to construct acceptors for regular languages. The structure of these acceptors corresponds to the parse tree of a regular expression defining the language. Solutions in this vein have been presented in [1, 2, 9]. Our approach has a number of appealing properties:

- the derivation is based on calculus: the design is formally derived from the specification, thereby ensuring the correctness of the solution;
- the design has a high degree of parallelism; in particular, concatenation is not implemented sequentially;
- in this design the empty string is recognized correctly (as opposed to [2]);
- in this design we have (as in [1]) a separate cell for concatenation;
- the design is relatively simple.

Furthermore, we show that our solution has constant response time [10], i.e. its response time does not depend on the length of the input stream.

The acceptor consists of cells that have the following characteristics (cf. [7]):

- each cell consumes streams of input values and produces streams of output values;
- each cell communicates with a fixed number of neighbor cells only;
- synchronization of cells is by message passing only;
- the communication behavior of a cell is independent of the values communicated.

The formalism we use to discuss the parallel computations is trace theory [4, 6, 8, 10]. Section 2 contains an overview of the theory and the notations needed for the derivation that is presented in Section 3.

The derivation method that is used can be found in [7, 10]. It is a calculational method: starting from the specification a solution is derived. In essence, it is predicate calculus that leads to the presented programs.

2. Parallel computations

A parallel computation is carried out by a network of cells that are interconnected by channels. A network, also called a *component*, communicates with its environment by sending and receiving messages.

The specification of a component consists of a relation between the sequences of values it receives and sends. This relation is called the input/output relation, or *i/o relation* for short, of the component.

For example, a component that repeatedly accepts two integers and outputs their sum may be specified by

input ports: a and b of type integer

output ports: c of type integer

i/o relation: $c(i) = a(i) + b(i), \quad i \geq 0$

Notice that the elements of sequences are indexed from 0 upwards. A component is described by a program text in a CSP-like notation [3]. A possible program for the component specified above is:

```
com adder(in a,b: int, out c: int):
    |[x,y: int;
      (a?x, b?y; c!(x+y))*
    ]|
moc
```

The first line of the program expresses that component *adder* has two input ports a and b of type integer and an output port c of type integer. In the second line, local variables x and y of type integer are introduced. The third line contains the *command*. Statement $a?x$ is an input statement, it is equivalent to $x := a(i)$ where i is the number of preceding communications along input port a. Statement $c!(x+y)$ is an output statement, it is equivalent to $c(j) := x+y$ where j is the number of preceding communications along output port c. The comma denotes that the order in which the surrounding statements are executed is irrelevant, in fact they may be executed in parallel. The comma has a higher binding power than the semicolon. Notice that in this case before each step of the repetition the number of preceding inputs along a and the number of preceding inputs along b equal the number of

preceding outputs along c. The combination $a?x, b?y; c!(x+y)$ establishes

$$x = a(i) \ \wedge \ y = b(i) \ \wedge \ c(i) = x + y$$

and, consequently, $c(i) = a(i) + b(i)$, as required by the i/o relation.

Component *adder* is *data-independent*: the order of communications does not depend on the values communicated. The *communication behavior* of component *adder* is $(a,b;c)^*$. In this paper all components will be data-independent. For a more formal treatment we refer to [7, 10].

Components may have previously defined *subcomponents*. As an example of a component with subcomponents, consider

```
com add3(in a,b,c : int, out d : int):
    sub p : adder bus
    |[x,y,z,w : int;
      a?x, b?y, c?z
    ; (p.a!(x+y), p.b!z; p.c?w
      ; a?x, b?y, c?z, d!w
      )*
    ]|
moc
```

Component add_3 has one subcomponent p of type *adder*. Ports $p.a$, $p.b$, and $p.c$ denote ports a, b, and c of subcomponent p. For the subcomponent we have, on account of the i/o relation of *adder*,

$$p.c(i) = p.a(i) + p.b(i)$$

The command of add_3 gives rise to an additional cell. That cell has $p.a$ and $p.b$ as output ports, and $p.c$ as input port, cf. Fig. 1. The command establishes

$$p.a(i) = a(i) + b(i)$$

$$p.b(i) = c(i)$$

$$d(i) = p.c(i)$$

Together with $p.c(i) = p.a(i) + p.b(i)$ this implies $d(i) = a(i) + b(i) + c(i)$. Hence, add_3 has i/o relation

$$d(i) = a(i) + b(i) + c(i)$$

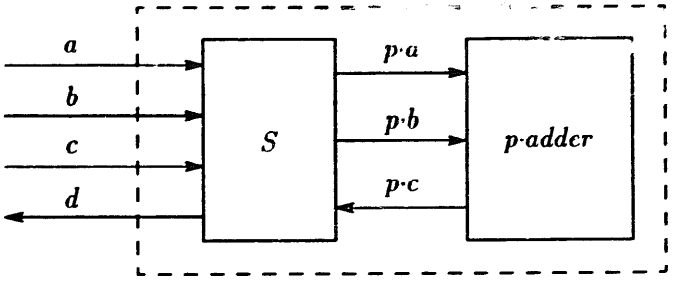

Fig. 1. Component add_3 with command S and subcomponent $p.adder$.

The communication behavior of the command is $a,b,c;(p.a,p.b;p.c;a,b,c,d)^*$. The *external* behavior, i.e. the communication behavior of add_3, is obtained by projection on the external ports: $a,b,c;(a,b,c,d)^*$.

We will use sequence functions (cf. [7] and [10]) to express the efficiency of programs. A sequence function is defined on the occurrences of communication events (the ith occurrence of a is denoted by (a, i)). If σ is a sequence function, then $\sigma(a, i)$ may be interpreted as the time slot in which the ith communication along a occurs. A sequence function should respect the (partial) order of the command of the component for which it is defined. A possible sequence function for the component in the above example is

$$\sigma(a, i) = \sigma(b, i) = \sigma(c, i) = 3i$$

$$\sigma(p.a, i) = \sigma(p.b, i) = 3i + 1$$

$$\sigma(p.c, i) = 3i + 2$$

$$\sigma(d, i) = 3i + 3$$

Of course, σ restricted to $\{p.a, p.b, p.c\}$ is valid sequence function for subcomponent p.

Notice that $\sigma(d, i) - \sigma(a, i) = 3$ and $\sigma(a, i+1) - \sigma(d, i) = 0$. The number of time slots between two consecutive external events is bounded by 3. We say that component add_3 has *constant response time.*

3. Acceptors for regular expressions

A regular expression E over an alphabet A of symbols defines a language $\mathcal{L}(E)$. Regular expressions and their languages are defined inductively by the following rules:

(i) ε is a regular expression,

$$\mathcal{L}(\varepsilon) = \{\varepsilon\}$$

where ε denotes the empty sequence;

(ii) c is a regular expression for $c \in A$,

$$\mathcal{L}(c) = \{c\}$$

(iii) if E and F are regular expressions, then $(E);(F)$ is a regular expression,

$$\mathcal{L}((E);(F)) = \mathcal{L}(E)\mathcal{L}(F) \quad (=\{uv \mid u \in \mathcal{L}(E) \wedge v \in \mathcal{L}(F)\})$$

(iv) if E and F are regular expressions then $(E)|(F)$ is a regular expression,

$$\mathcal{L}((E)|(F)) = \mathcal{L}(E) \cup \mathcal{L}(F)$$

(v) if E is a regular expression, then $(E)^*$ is a regular expression,

$$\mathscr{L}((E)^*) = \mathscr{L}(E)^*$$

where $\mathscr{L}(E)^*$ denotes the set of all finite concatenations of elements of $\mathscr{L}(E)$.

As usual, parentheses in regular expressions may be omitted by defining a priority for the operators. In order of decreasing priority we have the star, the semicolon, and the bar. For instance, $(c;d|e)^*$ denotes regular expression $(((c);(d))|(e))^*$. Instead of $\mathscr{L}(E)$ we will write E wherever this does not lead to any confusion.

An *acceptor* for regular expression E is a component that upon receiving a sequence of symbols determines whether that sequence belongs to E. We will solve the following problem: given regular expression E, construct component acc_E specified by (sequence $a(j: p \leq j < q)$ is denoted by $a[p..q)$):

input port: a of type symbol

output port: b of type boolean (SPEC$_0$)

i/o relation: $b(i) = a[0..i) \in E, \quad i \geq 0$

Component acc_E will be defined by induction on the structure of E. $E = \varepsilon$ yields, according to the definition of ε:

$$b(0) = \text{true}$$

$$b(i+1) = \text{false}, \quad i \geq 0$$

$E = c$ yields, according to the definition of c:

$$b(0) = \text{false}, \qquad b(1) = (a(0) = c)$$

$$b(i+2) = \text{false}, \quad i \geq 0$$

Choosing $(b;a)^*$ as external communication behavior these relations lead to the following programs:

```
com acc_ε(in a: sym, out b: bool):
    |[x: sym;
      b!true; (a?x; b!false)*
    ]|
moc

com acc_c(in a: sym, out b: bool):
    |[x: sym;
      b!false; a?x; b!(x = c)
    ; (a?x; b!false)*
    ]|
moc
```

Next, the semicolon is considered. Let $E;F$ be a regular expression. We assume the existence of subcomponent p of type acc_E and subcomponent q of type acc_F.

For $i \geq 0$ we derive (denoting the existential quantifier by **E**):

$$
\begin{array}{ll}
 & b(i) \\
= & \{\mathrm{SPEC}_0\} \\
 & a[0..i) \in E\,;F \\
= & \{\text{definition of } E\,;F\} \\
 & (\mathbf{E}k : 0 \leq k \leq i : a[0..k) \in E \wedge a[k..i) \in F) \\
= & \{\text{choose } p.a = a, \mathrm{SPEC}_0\} \\
 & (\mathbf{E}k : 0 \leq k \leq i : p.b(k) \wedge a[k..i) \in F) \\
= & \{\text{split off } k = i, \mathrm{SPEC}_0\} \\
 & (\mathbf{E}k : 0 \leq k < i : p.b(k) \wedge a[k..i) \in F) \vee (p.b(i) \wedge q.b(0))
\end{array}
$$

The last expression cannot be simplified any further. However, we can extend the original i/o relation in such a way that its shape resembles the first disjunct. We do so by introducing an auxiliary input port e of type boolean. If we were to choose

$$b(i) = (\mathbf{E}k : 0 \leq k < i : e(k) \wedge a[k..i) \in E), \quad i \geq 0$$

as i/o relation, component acc_E could not be used to determine whether $\varepsilon \in E$. Therefore, we generalize specification SPEC_0 to

$$
\begin{array}{l}
b(0) = \varepsilon \in E \\
b(i) = (\mathbf{E}k : 0 \leq k < i : e(k) \wedge a[k..i) \in E), \quad i \geq 1.
\end{array}
\qquad (\mathrm{SPEC}_1)
$$

Then feeding the (extended) component with input $e(0) = \text{true}$ and $e(i+1) = \text{false}$ $(i \geq 0)$ results in the effect specified by SPEC_0. The following derivations show that starting from the above specification one can indeed derive a component for acc_E defined inductively on the structure of E. $E = \varepsilon$ now yields:

$$b(0) = \text{true}$$

$$b(i+1) = \text{false}, \quad i \geq 0$$

$E = c$ yields:

$$b(0) = \text{false}$$

and for $i \geq 0$:

$$
\begin{array}{ll}
 & b(i+1) \\
= & \{(\mathrm{SPEC}_1)\} \\
 & (\mathbf{E}k : 0 \leq k < i+1 : e(k) \wedge a[k..i+1) = c) \\
= & \{\text{calculus}\} \\
 & e(i) \wedge (a(i) = c)
\end{array}
$$

Choosing $(b\,;e,a)^*$ as external communication behavior these relations lead to the

following programs:

```
com acc_t (in a : sym, e : bool, out b : bool):
    [[ x : sym ; w : bool;
       b!true ; (e?w, a?x ; b!false)*
    ]]
moc

com acc_f (in a : sym, e : bool, out b : bool):
    [[ x : sym ; w : bool;
       b!false ; (e?w, a?x ; b!(w ∧ (x = c)))*
    ]]
moc
```

Let $E ; F$ be a regular expression. Let p and q be subcomponents of type acc_t and acc_f, respectively. Then:

$$
\begin{aligned}
& b(0) \\
= \quad & \{\text{SPEC}_1\} \\
& \varepsilon \in E ; F \\
= \quad & \{\text{definition of } E ; F\} \\
& \varepsilon \in E \ \wedge \ \varepsilon \in F \\
= \quad & \{\text{SPEC}_1\} \\
& p.b(0) \ \wedge \ q.b(0)
\end{aligned}
$$

For $i \geq 1$ we derive:

$$
\begin{aligned}
& b(i) \\
= \quad & \{\text{SPEC}_1\} \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in E ; F) \\
= \quad & \{\text{definition of } E ; F\} \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ (\mathbf{E}j : k \leq j \leq i : a[k..j) \in E \ \wedge \ a[j..i) \in F)) \\
= \quad & \{\text{predicate calculus}\} \\
& (\mathbf{E}j : 0 \leq j \leq i : (\mathbf{E}k : 0 \leq k \leq j \ \wedge \ k < i : e(k) \ \wedge \ a[k..j) \in E) \ \wedge \ a[j..i) \in F) \\
= \quad & \{\text{split off } j = i\} \\
& (\mathbf{E}j : 0 \leq j < i : (\mathbf{E}k : 0 \leq k \leq j : e(k) \ \wedge \ a[k..j) \in E) \ \wedge \ a[j..i) \in F) \ \vee \\
& ((\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in E) \ \wedge \ \varepsilon \in F) \\
= \quad & \{\text{split off } k = j, \text{ choose } p.e = e \text{ and } p.a = a, \text{ SPEC}_1\} \\
& (\mathbf{E}j : 0 \leq j < i : ((\mathbf{E}k : 0 \leq k < j : e(k) \ \wedge \ a[k..j) \in E) \ \vee \\
& \qquad (e(j) \ \wedge \ \varepsilon \in E)) \ \wedge \ a[j..i) \in F) \ \vee \\
& (p.b(i) \ \wedge \ q.b(0)) \\
= \quad & \{p.e = e, \ p.a = a, \ \text{SPEC}_1\} \\
& (\mathbf{E}j : 0 \leq j < i : ((j > 0 \ \wedge \ p.b(j)) \ \vee \ (e(j) \ \wedge \ p.b(0))) \ \wedge \ a[j..i) \in F) \ \vee \\
& (p.b(i) \ \wedge \ q.b(0)) \\
= \quad & \{\text{choose } q.e(l) = (l > 0 \ \wedge \ p.b(l)) \ \vee \ (e(l) \ \wedge \ p.b(0)), \ q.a = a, \ \text{SPEC}_1\} \\
& q.b(i) \ \vee \ (p.b(i) \ \wedge \ q.b(0))
\end{aligned}
$$

We recapitulate the relations found

$$p.a(i) = q.a(i) = a(i), \quad i \geq 0$$

$$p.e(i) = e(i), \quad i \geq 0$$

$$q.e(0) = e(0) \wedge p.b(0)$$

$$q.e(i) = p.b(i) \vee (e(i) \wedge p.b(0)), \quad i \geq 1$$

$$b(0) = p.b(0) \wedge q.b(0)$$

$$b(i) = (p.b(i) \wedge q.b(0)) \vee q.b(i), \quad i \geq 1$$

The program of $acc_{E;F}$ contains six variables, satisfying the following relations:

$$x = a(i), \quad w = e(i), \qquad i \geq 0$$

$$p0 = p.b(0), \qquad q0 = q.b(0)$$

$$pb = p.b(i), \quad qb = q.b(i), \qquad i \geq 1$$

Notice that the initialization $pb := \text{false}$ removes the difference in shape between $q.e(0)$ and $q.e(i)$, $i \geq 1$. We obtain the following program text.

```
com acc_{E;F}(in a : sym, e : bool, out b : bool):
    sub p : acc_E, q : acc_F bus
    |[x : sym ; w,pb,qb,p0,q0 : bool;
      p.b?p0, q.b?q0; b!(p0 ∧ q0), pb := false
    ; (e?w, a?x
      ; p.e!w, q.e!((w ∧ p0) ∨ pb), p.a!x, q.a!x
      ; p.b?pb, q.b?qb
      ; b!((pb ∧ q0) ∨ qb)
      )*
    ]|
moc
```

For the bar we follow the same strategy (although it turns out to be much simpler). Let $E \mid F$ be a regular expression. Let p and q be subsomponents of type acc_E and acc_F, respectively. Then:

$$\begin{aligned}
& b(0) \\
= \quad & \{\text{SPEC}_1\} \\
& \varepsilon \in E \mid F \\
= \quad & \{\text{definition of } E \mid F\} \\
& \varepsilon \in E \ \vee \ \varepsilon \in F \\
= \quad & \{\text{SPEC}_1\} \\
& p.b(0) \ \vee \ q.b(0)
\end{aligned}$$

and for $i \geq 1$:

$$
\begin{aligned}
& b(i) \\
= \quad & \{\text{SPEC}_1\} \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in E \,|\, F) \\
= \quad & \{\text{definition of } E \,|\, F\} \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ (a[k..i) \in E \ \vee \ a[k..i) \in F)) \\
= \quad & \{\text{predicate calculus}\} \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in E) \ \vee \\
& (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in F) \\
= \quad & \{\text{choose } p.a = a,\ q.a = a,\ p.e = e,\ q.e = e\} \\
& p.b(i) \ \vee \ q.b(i)
\end{aligned}
$$

Thus, we have for $i \geq 0$:

$$p.a(i) = q.a(i) = a(i)$$

$$p.e(i) = q.e(i) = e(i)$$

$$b(i) = p.b(i) \ \vee \ q.b(i)$$

From these relations a program is easily derived:

```
com acc_{E|F}(in a : sym, e : bool, out b : bool) :
    sub p : acc_E, q : acc_F bus
    |[x : sym ; w,pb,qb : bool;
      ( p.b?pb, q.b?qb
      ; b!(pb ∨ qb)
      ; e?w, a?x
      ; p.e!w, q.e!w, p.a!x, q.a!x
      )*
    ]|
moc
```

Finally, we have to construct acc_{E^*}. We assume the existence of a subcomponent p of type acc_E. Then:

$$
\begin{aligned}
& b(0) \\
= \quad & \{\text{SPEC}_1\} \\
& \varepsilon \in E^* \\
= \quad & \{\text{definition of } E^*\} \\
& \text{true}
\end{aligned}
$$

and for $i \geq 1$:

$$
\begin{array}{ll}
 & b(i) \\
= & \{\text{SPEC}_1\} \\
 & (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ a[k..i) \in E^*) \\
= & \{\mathcal{S}(E^*)\backslash\{\varepsilon\} = \mathcal{S}(E^*)(\mathcal{S}(E)\backslash\{\varepsilon\})\} \\
 & (\mathbf{E}k : 0 \leq k < i : e(k) \ \wedge \ (\mathbf{E}j : k \leq j < i : a[k..j) \in E^* \ \wedge \ a[j..i) \in E)) \\
= & \{\text{predicate calculus}\} \\
 & (\mathbf{E}j : 0 \leq j < i : (\mathbf{E}k : 0 \leq k \leq j : e(k) \ \wedge \ a[k..j) \in E^*) \ \wedge \ a[j..i) \in E) \\
= & \{\text{split off } k = j,\ \varepsilon \in E^*\} \\
 & (\mathbf{E}j : 0 \leq j < i : ((\mathbf{E}k : 0 \leq k < j : e(k) \ \wedge \ a[k..j) \in E^*) \ \vee \ e(j)) \ \wedge \ a[j..i) \in E) \\
= & \{\text{split off } j = 0,\ \text{predicate calculus, SPEC}_1\} \\
 & (e(0) \ \wedge \ a[0..i) \in E) \ \vee \\
 & (\mathbf{E}j : 0 < j < i : (e(j) \ \vee \ b(j)) \ \wedge \ a[j..i) \in E) \\
= & \{\text{choose } p.a = a,\ p.e(0) = e(0),\ p.e(l) = e(l) \ \vee \ b(l)\ (l \geq 1),\ \text{SPEC}_1\} \\
 & p.b(i)
\end{array}
$$

We recapitulate the relations found:

$$p.a(i) = a(i), \quad i \geq 0$$

$$p.e(0) = e(0)$$

$$p.e(i) = e(i) \ \vee \ p.b(i), \quad i \geq 1$$

$$b(0) = \text{true}$$

$$b(i) = p.b(i), \quad i \geq 1$$

A program for $acc_{t\cdot}$ is given below:

```
com acc_t·(in a : sym, e : bool, out b : bool):
      sub p : acc_t bus
      |[x : sym ; w,pb : bool;
        p.b?pb ; b!true , pb := false
      ; (e?w , a?x
        ; p.a!x , p.e!(w ∨ pb)
        ; p.b?pb
        ; b!pb
        )*
      ]|
moc
```

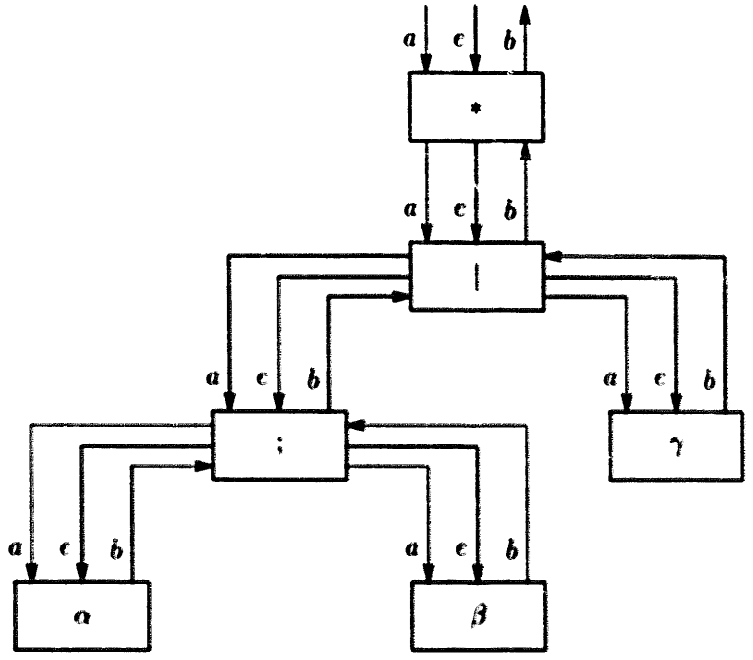

Fig. 2. An acceptor for $(\alpha;\beta|\gamma)^*$.

This concludes the derivation of acc_E. Figure 2 shows an acceptor for the expression $(\alpha;\beta|\gamma)^*$. The boxes denote the cells of the corresponding components.

4. Analysis of the solution

An acceptor has the form of a tree. The leaves of the tree are either an acc_ε or an acc_c. The depth $d(E)$ of the tree of acc_E is defined inductively by:

$$d(\varepsilon)=0$$

$$d(c)=0$$

$$d((E)|(F))=(d(E)\ \mathbf{max}\ d(F))+1$$

$$d((E);(F))=(d(E)\ \mathbf{max}\ d(F))+1$$

$$d((E)^*)=d(E)+1$$

The communication behaviors of the cells are as follows.

acc_ε and acc_c: $(b;e,a)^*$

$acc_{E;F}$ and $acc_{E|F}$: $(p.b,q.b;b;e,a;p.e,p.a,q.e,q.a)^*$

acc_{E^*}: $(p.b;b;e,a;p.e,p.a)^*$

Notice that the external behavior of all components is $(b;e,a)^*$, and their internal behavior is $(p.b;p.e,p.a)^*$. This guarantees the absence of deadlock [10]. Since a and e may always happen concurrently we refrain from mentioning e in the sequel.

A similar observation holds for $p.s$ and $q.s$ where s is a, b, or e. Hence, the behavior of the leaves is characterized by $(b;a)^*$ and the behavior of the nodes by $(p.b;b;a;p.a)^*$.

A possible sequence function for the external behavior that respects the behaviors of all subcomponents is given by:

$$\sigma_{acc_t}(b,i) = 2\cdot(1+d(E))\cdot i + d(E)$$

$$\sigma_{acc_t}(a,i) = 2\cdot(1+d(E))\cdot i + 1 + d(E)$$

which can be proved by induction on the structure of E. Hence, the interval between two consecutive external communications is at most $2\cdot d(E)+1$, which is a measure for the response time of component acc_t. Due to the parallelism the response time does not depend on the length of the input: component acc_t has constant response time. The time needed to recognize a sentence is linearly dependent on its length.

Component acc_t is reusable: let c_0 be a symbol not occurring in E and let $N \geq 0$. If

$$a(N) = c_0$$

then

$$b(0) = \varepsilon \in E$$

$$b(i) = (\mathbf{E}k : 0 \leq k < i : e(k) \wedge a[k..i) \in E), \quad 1 \leq i \leq N$$

$$b(N+1) = \text{false}$$

$$b(N+1+i) = (\mathbf{E}k : 0 \leq k < i : e(N+1+k) \wedge a[N+1+k..N+1+i) \in E), \quad i \geq 1$$

Notice that for cells corresponding to a star, semicolon, or bar, outputs b, $p.e$, and $q.e$ do not depend on input a. These cells merely pass on the symbols received via a to their subcells as is reflected by relations $p.a(i) = a(i)$ and $q.a(i) = a(i)$. For cells corresponding to ε output b does not depend on input a either. These cells discard the symbols received via a. Cells corresponding to a symbol are the only cells that do inspect the symbols received via a. They do not pass the symbols on. Hence, instead of passing input a to these cells via the entire tree one may feed it to these cells directly. Notice that the cells corresponding to a symbol are leaf cells in the tree.

5. Concluding remarks

We have shown how a difficult problem can be solved in a systematic way. The derivation of the solution illustrates programming techniques that are applicable in the field of parallel programming. The idea of introducing auxiliary channel e is comparable with the introduction of program variables in sequential programming.

The introduction of channel e is based on formal derivations. A different generalization of the i/o relation is

$$b(i) = (\mathbf{E}k : 0 \leq k \leq i : e(k) \ \wedge \ a[k..i) \in E), \quad i \geq 0$$

Then, however, $b(0) = e(0) \wedge \varepsilon \in E$, making $b(0)$ dependent on the value of an input. It turns out that this yields unsolvable complications in the derivation of acc_{E^*}.

The structure of the design reflects the structure of the problem. Properties of the solution (e.g. response time) can be proved using structural induction.

The ultimate solution is efficient: the time needed to recognize a sentence is linearly dependent on its length. The cells are relatively simple and they are easily implemented as circuits. Communication by message passing may be implemented by a handshaking protocol as described in [5].

Acknowledgement

We are very grateful to various members of the Computing Science Group at the Eindhoven University of Technology. Especially W.H.J. Feijen, Rob Hoogerwoord and Martin Rem have been very helpful with their suggestions on an earlier draft of this paper. Also, Roland Backhouse made some helpful remarks. Finally, we would like to thank an anonymous referee for his scrupulous reading of the text.

References

[1] T.S. Anantharaman, E.M. Clarke, M.J. Foster and B. Mishra, Compiling path expressions into VLSI circuits, *Distributed Comput.* **1** (1986) 150-160.

[2] M.J. Foster and H.T. Kung, Recognize regular languages with programmable building-blocks, in: J.P. Gray, ed., *VLSI 81: Very Large Scale Integration* (Academic Press, London, 1981) 75-84.

[3] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (1978) 666-677.

[4] A. Kaldewaij, A formalism for concurrent processes, Ph.D. Thesis, Eindhoven University of Technology (1986).

[5] A. Kaldewaij, The translation of processes into circuits, in: J.W. de Bakker, A.J. Nijman and P.C. Treleaven, eds., *PARLE: Parallel Architectures and Languages Europe, I: Parallel Architectures*, Lecture Notes in Computer Science **258** (Springer, Berlin, 1987) 195-212.

[6] M. Rem, Concurrent computations and VLSI circuits, in: M. Broy, ed., *Control Flow and Data Flow: Concepts of Distributed Programming* (Springer, Berlin, 1985) 399-437.

[7] M. Rem, Trace theory and systolic computations, in: J.W. de Bakker, A.J. Nijman and P.C. Treleaven, eds., *PARLE: Parallel Architectures and Languages Europe, I: Parallel Architectures*, Lecture Notes in Computer Science **258** (Springer, Berlin, 1987) 14-33.

[8] J.L.A. van de Snepscheut, *Trace Theory and VLSI Design*, Lecture Notes in Computer Science **200** (Springer, Berlin, 1985).

[9] H. van de Wetering, Acceptors of regular sets, Master's Thesis, Eindhoven University of Technology (1985).

[10] G. Zwaan, Parallel computations, Ph.D. Thesis, Eindhoven University of Technology (1989).